Parallel Performance for a Real Time Lattice Boltzmann Code

Mark A. Woodgate[a,1], George N. Barakos[a,2], Rene Steijl[a,3], Gavin J. Pringle[b,4]

[a] *The University of Glasgow, CFD Laboratory, School of Engineering, James Watt South Building, G12 8QQ, U.K.*
[b] *EPCC, University of Edinburgh, James Clerk Maxwell Building, Edinburgh, EH9 3FD, U.K.*

Abstract

This paper details a real time Lattice Boltzmann solver for computing unsteady wakes. The formulation of the lattice Boltzmann method is first presented followed by a discussion of boundary conditions. This is followed by the details of how the basic algorithm was improved to increase single-core efficiency. A discussion of high Reynolds number implementations, and their effect on the overall computational cost follows, and examples are presented alongside performance data on a variety of CPUs.

Keywords: Real-Time, Lattice Boltzmann Method, Wake

1. Introduction

Computational fluid dynamic methods have become increasingly sophisticated and accurate over the past 20 years. However, they are orders or

[1]Research Associate, Mark.Woodgate@glasgow.ac.uk
[2]Professor, George.Barakos@glasgow.ac.uk
[3]Lecturer, Rene.Steijl@glasgow.ac.uk
[4]Applications Consultant, gavin@epcc.ed.ac.uk

magnitude too slow for real time flow computation and so, analytical or simplified aerodynamic models are still used.

Regarding wakes, there are a number of methods to represent these in real-time flight simulations. Some methods use an analytical model or a set of velocity vectors in tabular format. Another method for real-time simulation is the use of a free wake model as presented by Horn et al. (2006) who performed a parametric study of the wake parameters to achieve real-time execution with minimal differences from a spatially and temporally converged response, which at the time did not achieve real-time execution. Lastly, a method suggested by Spence et al. (2007) used implicit large eddy simulations (ILES) to build a database which was accessed in real time. This was achieved through the use of a data compression scheme, mesh simplifications and of kd-trees for fast data queries. Recently, Khan et al. (2015) demonstrated the use of the Lattice Boltzmann Method (LBM), implemented on a Graphics Processor Unit (GPU), for real-time simulations of flows in indoor environments. The LBM is developed from the Lattice Gas Automata (LGA) method by Frisch et al. (1986) and is an explicit discretization Boltzmann's equation. The method is both parallel and efficient, see Chen and Doolen (1998), due to only using local operations and is a leading candidate for real-time simulations.

The goal of the current work is to explore the possibility of using real-time Computational Fluid Dynamics (CFD) methods for problems with low speeds, and weak compressibility effects. These problems include the wakes of wind turbines, buildings and ships where typical wind speeds are between 10 to 20 m/s. The ultimate aim is to use this fast CFD method to account

for the effects of the vortical wakes in real-time flight simulation.

2. The Lattice Boltzmann Method

The LBM uses a simplified kinetic model which includes the essential microscopic effects to calculate the macroscopic averaged quantities of the Navier-Stokes equations. This is achieved by solving the discrete-velocity Boltzmann equation. A regular lattice is used over the computational domain and a particle distribution function represent the probability of a particle having a given velocity at each lattice point. The movement of the particles is restricted by a subset of neighbouring lattice points. The discrete collision rule is finally replaced by an approximate collision operator with the Bhatnagar-Gross-Krook (BGK) model being the most widely used (see Chen and Doolen (1998)). A common labelling for the lattices in the LBM is DdQq, where d is the spatial dimension and q are the number of microscopic velocities. Some common three dimensional lattice constructions used for fluid flows are D3Q15, D3Q19 and D3Q27 as shown in figure 1. The D3Q19 model has been chosen in this work to keep the computational cost low while maintaining an isotropic lattice.

The LBM numerical solution involves two steps. First, there is a collision step where

$$f_i^t(\bar{x}, t+\delta t) = f_i(\bar{x}, t) + \frac{1}{\tau_f}\left[f_i^{eq}(\rho, \bar{u}) - f_i(\bar{x}, t)\right] = (1-\frac{1}{\tau_f}) f_i(\bar{x}, t) + \frac{1}{\tau_f} f_i^{eq}(\rho, \bar{u}). \tag{1}$$

where f_i represents the particle distribution function, which is the fraction of particles located at position $\bar{x}$ at time t moving at microscopic velocity $\bar{e}_i$, and i are discrete directions of momentum which are the q chosen collocation

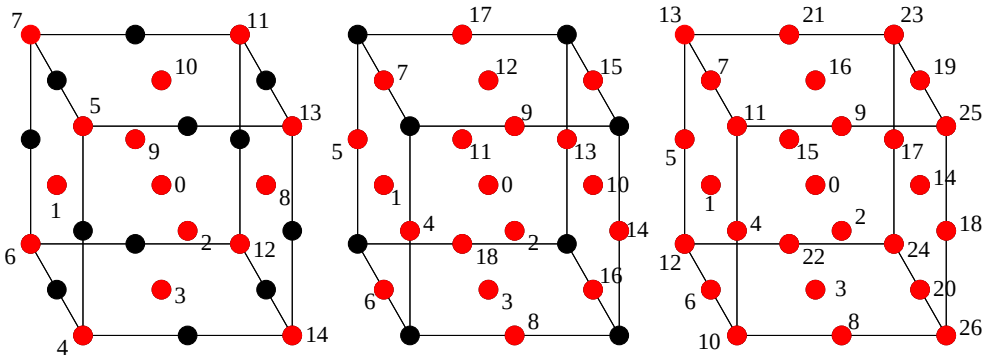


Fig. 1 – Common three dimensional lattices with the indices re-ordered to minimize the traversal of computer memory, and hence reduce the memory bandwidth requirements of the the LBM as discussed in section 3.1.7

points of the discrete-velocity Boltzmann equation and determine the basic structure of the LBM grid.

This is then followed by a streaming step where the value of $f_i^t(\bar{x}, t + \delta t)$ is shifted in space along the lattice velocity $\bar{e}_i$,

$$f_i(\bar{x} + c\bar{e}_i\delta t, t + \delta t) = f_i^t(\bar{x}, t + \delta t), \tag{2}$$

where c is the lattice speed. The relaxation time τ determines how fast the equilibrium position is approached and is also related to the kinematic viscosity of the fluid. The equilibrium state $f_i^{eq}(\rho, \bar{u})$ is a low Mach number approximation of the Maxwell-Boltzmann equilibrium distribution function, where ρ is the the macroscopic value of the density and $\bar{u}$ is the macroscopic value of the velocity.

The density ρ and the velocity $\bar{u}$ are obtained from the zeroth and first

moments of the distribution functions

$$\rho = \sum_{i=0}^{18} f_i, \qquad \rho\bar{u} = \sum_{i=0}^{18} c\bar{e}_i f_i, \tag{3}$$

with the discrete velocity set $\bar{e}_i$ defined as:

$$\bar{e}_i = \begin{cases} (0,0,0) & i = 0 & w_i = 1/3 \\ (\pm 1,0,0),(0,\pm 1,0),(0,0,\pm 1) & i = 1-6 & w_i = 1/18 \\ (\pm 1,\pm 1,0) & i = 7-10 & w_i = 1/36 \\ (\pm 1,0,\pm 1) & i = 11-14 & w_i = 1/36 \\ (0,\pm 1,\pm 1) & i = 15-18 & w_i = 1/36 \end{cases} . \tag{4}$$

The equilibrium state is calculated by

$$f_i^{eq}(\rho,\bar{u}) = \rho w_i \left(1 + \frac{3\bar{e}_i \cdot \bar{u}}{c^2} + \frac{9(\bar{e}_i \cdot \bar{u})^2}{2c^4} - \frac{3\bar{u}^2}{2c^2}\right) \tag{5}$$

where the w_i are the weight co-efficients defined in equation 4.

It can be shown through a Chapman-Enskog expansion (see Chapman and Cowling (1991)) that the Navier-Stokes equations can be obtained from the lattice BGK model. First, by using a 2nd order Taylor series expansion about the left hand side of equation 1 the particle distribution function is split into equilibrium and non equilibrium components. After, using the Chapman-Enskog expansion, which expands the non equilibrium part in a power series of the Knudsen number, the Taylor series can be decomposed into different orders of magnitude of the Knudsen number to obtain the continuum equations which recover the Navier-Stokes equation assuming small density variations. The Knudsen number ($Kn = \lambda/L$) is a dimensionless number defined as the ratio of the molecular mean free path length (λ) to a representative physical length scale. This length scale (L) could be, for example, the chord of an aerofoil.

2.1. Viscosity and Time-step

The relaxation time τ is related to the viscosity of the fluid. If the lattice speed c is $\Delta x/\Delta t$ then the kinematic viscosity of the fluid, ν, is given by

$$\nu = \frac{(2\tau - 1)}{6}\frac{(\Delta x)^2}{\Delta t}, \tag{6}$$

or

$$\tau = 0.5 + 3\nu_{lb} = 0.5 + u_{lb}(N-1)/Re \tag{7}$$

where ν_{lb} and u_{lb} are the viscosity and speed in lattice units with Re the Reynolds number. However, as τ approaches $1/2$ the scheme becomes unstable as the lattice viscosity is too low to dissipate the shortest wavelengths of errors. The non positivity of the distribution function is the reason for the numerical instability of the LBM, see Succi et al. (2002), for example, for the role of the H theorem in enforcing the method to comply with the second law of thermodynamics, as needed for a numerically stable scheme.

Under testing, the best case was found to be a $\tau = 0.5008$ for periodic domain boundaries. This means the Reynolds number is of the order of thousands and so, at least two orders of magnitude too small for wake computations. In practice, the true range of Reynolds numbers is much narrower due to instabilities introduced by complex geometries and boundary conditions.

2.2. High Reynolds Number Formulations

The stability of the LBM can be enhanced for high Reynolds numbers. Most of the schemes outlined below do this by using a different value of τ at each lattice point.

2.2.1. Entropic Lattice Boltzmann Method

The Reynolds number can be increased by several orders of magnitude using the Entropic Lattice Boltzmann method of Chikatamarla et al. (2006), and Karlin et al. (1999) which allows the Lattice Boltzmann models to support a discrete H-theorem through the use of a modified equilibrium distribution function:

$$f_i^{eq} = \rho w_i \sum_{\alpha=1}^{3} \left(2 - \sqrt{1+3u_\alpha^2}\right) \left(\frac{2u_\alpha + \sqrt{1+3u_\alpha^2}}{1-u_\alpha}\right)^{e_{i\alpha}}, \tag{8}$$

where $e_{i\alpha}$ is the α component of $\bar{e}_i$ given in equation 4. The relaxation process is also modified using an adjustable parameter β at every simulation step by means of the solution of the h-function monotonicity constraint

$$H(f) = H(f - \beta(f - f^{eq})) \tag{9}$$

where

$$H(f) = \sum_{i=1}^{q} f_i \ln\left(\frac{f_i}{w_i}\right). \tag{10}$$

This produces an unconditionally stable numerical scheme via local adjustments to the relaxation time via the parameter $2/\beta$. When close to equilibrium $\beta = 2$, and the original scheme is recovered.

2.2.2. Smagorinsky Subgrid Model

A sub-grid scale closure model is widely used in the numerical simulation of turbulent flows with the lattice Boltzmann method. Malaspinas and Sagaut (2012) proposed a consistent way of including sub-grid closure models in the BGK Boltzmann equation for large eddy simulations of turbulent flows. The used a Hermite expansion of the velocity distribution function

and showed a connection between the new models and the current standard practice, proving that a single, modified, scalar relaxation time to account for the sub-grid viscosity is not consistent for compressible cases.

2.3. Boundary conditions

The boundary conditions for the lattice Boltzmann methods are quite different from the finite volume method. The difference is that boundary conditions, in the control finite volume method, are applied via the physical state of the macroscopic variables, e.g. zero wall velocity, while no such meaning can be applied to the velocity distribution function on the boundary with the LBM. Hence the velocities distribution function must be modified to apply boundary conditions. There is no unique way of doing this, and the way boundary conditions are implemented affects not only the numerical accuracy but also the stability of the computations.

There are two approaches: *wet boundary* conditions where the nodes are treated as still in the fluid but infinitesimally close to the wall, and *bounce-back boundary* conditions that place the wall half-way between the boundary nodes and the first row of fluid cells. For wet boundary conditions, the collision step is applied to the boundary nodes because they are inside the fluid domain while for bounce-back boundary conditions this is not the case since the nodes are not within the fluid domain.

2.3.1. Bounce-back boundary conditions

These boundary conditions are used to for both slip/symmetry and no-slip wall boundary conditions. In this case, when the distribution function streaming reaches the boundary node it will scatter back into the fluid. The

two boundary types are implemented by changing the direction in which the distribution function is scattered.

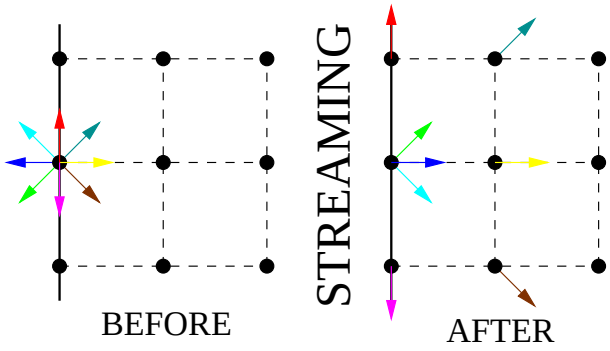
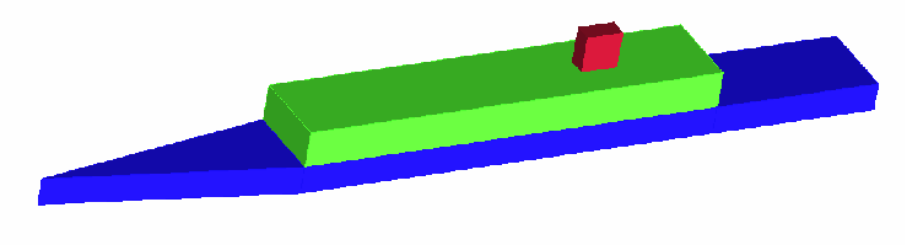


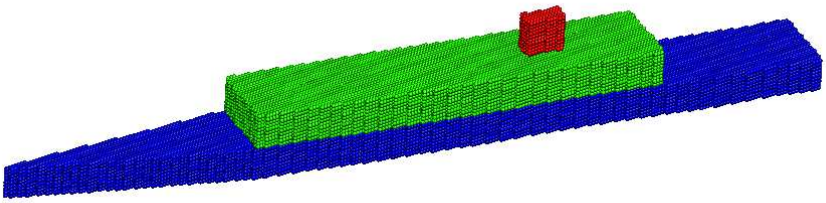
Fig. 2 – The distribution function for a boundary node for the full bounce-back condition before and after streaming.

For the full bounce-back condition, the incoming directions of the distribution function are reversed when they hit a boundary node and this process does not require knowledge of the orientation of the boundary. So complex geometries require no extra computation. The streaming of the full bounce-back can be seen in figure 2. This boundary condition acts half-way between nodes and not at boundary nodes. For a general geometry, the lattice points inside a solid need to be flagged as such. Both a lattice and a STereoLithography (STL) file of the geometry are needed, and a simple utility code can be written to return all the lattice points contained inside the geometry. An example of this can be seen for the Simple Frigate Shape 2 (SFS2) used by Crozon et al. (2014) in figure 3. Since the geometry has been rotated by 15 degrees none of the surfaces align with the lattice an so a "staircase" formation is obtained on every surface. At the current resolution there are just

enough points to resolve the stack on the superstructure of the SFS2.



(a) STereoLithography file of SFS2.



(b) Flagged lattice points for SFS2 after rotation by 15 degrees.

Fig. 3 – Flagging of bounce-back lattice points for an STL geometry.

2.3.2. Zou-He velocity and pressure boundary conditions

This boundary condition is used to model flows with a prescribed velocity or pressure/density at the boundary. It was originally introduced by Zou and He (1997). They used the relationship between the distribution functions and the prescribed velocity or density to solve for the unknown distribution functions after streaming. However, this does not provide enough equations and so the extra assumption made by Zou and He was that the bounce-back rule still holds for the non-equilibrium part of the particle distribution normal

to the boundary. This means that the boundary condition depends on the boundary orientation and is hard to generalize for complex geometries.

2.3.3. Moving wall boundary conditions

If a solid boundary moves, lattice nodes will cross the solid/fluid interface and change form interior to exterior nodes, and vice versa. In general, the numbers of fluid and solid nodes are not conserved over time. When a lattice point moves from fluid to solid the node is removed and its momentum is transferred to the solid. The creation of a fluid lattice point requires initialization of the unknown part of the distribution function as well as the density and velocity. The simplest method is to estimate the density with the average of neighbouring lattice points, set the velocity equal to the velocity of the body, and initialize the unknown distribution functions with their equilibrium values. Finally, the linear and angular momentums of the new fluid lattice point need to be removed from the solid. This method has two drawbacks, namely the total mass of the system is not generally conserved, and the non-equilibrium part of the new lattice node is also missing which may lead to flow field distortions.

3. Numerical Implementation of Lattice Boltzmann schemes

In the following section we discuss the implementation of the LBM. Two different situations will be discussed looking at the computational cost and at the amount of memory required per lattice point.

3.1. Basic Implementation of the LBM

The starting point is a basic LB code with $3Q$ variables per lattice node. The code is split into three steps with 4 loops in total. The first loop calculates the macroscopic density ρ, the second calculates the macroscopic velocity field, the third is the collision step, and the fourth is the streaming step. The flow case considered, corresponds to a periodic domain in all directions and so particles streamed to the n_x index need to be copied into 0 etc. This is achieved through the use of the modulo operation (lines 53,54 and 55 in Listing 1). Note that the width of the array was added to the modulo to avoid out of bounds errors.

The following listing presents the four loops where the lattice is of size n_x by n_y by n_z. The density is stored in the three-dimensional array `rho` while the three components of the velocity are stored in `ux`, `uy` and `uz`. All real arrays and matrices are stored as doubles. The three components of $\bar{e}_i$ of equation 4 are stored in `ex`, `ey` and `ez`, while `cx`, `cy` and `cz` are the integer counterparts used for array offsets, and the weights w_i are stored in `w`. `omega` is $1/\tau$ of equation 7. There are two sets of distribution functions `Fin` (before collision) and `Fout` (after collision) as well as the equilibrium distribution function `feq` of equation 5. All other variables are loop counters used for intermediate calculations.

Listing 1 – Basic Implementation

```
1 /* 1st Loop Calculate the macroscopic values of rho */
2 for i = 0 to n_x - 1 do
3   for j = 0 to n_y - 1 do
4     for k = 0 to n_z - 1 do
```

```
5         Tmp = 0.0
6         for l = 1 to q-1 do
7           Tmp = Tmp + Fin[i][j][k][l]
8         end
9         rho[i][j][k] = Tmp
10      end
11    end
12  end
13  /* 2nd Loop Calculate the macroscopic values of u,v,w */
14  for i = 0 to n_x - 1 do
15    for j = 0 to n_y - 1 do
16      for k = 0 to n_z - 1 do
17        Tmpu = 0.0
18        Tmpv = 0.0
19        Tmpw = 0.0
20        for l = 0 to q - 1 do
21          Tmpu = Tmpu + Fin[i][j][k][l]*ex[l]
22          Tmpv = Tmpv + Fin[i][j][k][l]*ey[l]
23          Tmpw = Tmpw + Fin[i][j][k][l]*ez[l]
24        end
25        ux[i][j][k] = Tmpu/rho[i][j][k]
26        uy[i][j][k] = Tmpv/rho[i][j][k]
27        uz[i][j][k] = Tmpw/rho[i][j][k]
28      end
29    end
```

```
30 end
31 /* 3rd Loop Collision Step */
32 for i = 0 to n_x - 1 do
33   for j = 0 to n_y - 1 do
34     for k = 0 to n_z - 1 do
35       v3 = ux[i][j][k]*ux[i][j][k] + uy[i][j][k]*uy[i][j][k]
36          + uz[i][j][k]*uz[i][j][k]
37       for l = 0 to q - 1 do
38         v1=ex[l]*ux[i][j][k]+ey[l]*uy[i][j][k]+ez[l]*uz[i][j][k]
39         v2=v1 * v1
40         feq[i][j][k][l] =rho[i][j][k] * w[l] *
41                          ( 1.0 + 3.0*v1 + 4.5*v2 - 1.5*v3)
42         Fout[i][j][k][l]=Fin[i][j][k][l]
43                          -omega*(Fin[i][j][k][l]-feq[i][j][k][l])
44       end
45     end
46   end
47 end
48 /* 4th Loop Streaming Step */
49 for i = 0 to n_x - 1 do
50   for j = 0 to n_y - 1 do
51     for k = 0 to n_z - 1 do
52       for l = 0 to q - 1 do
53         iX = (i + n_x + cx[l])%n_x
54         iY = (j + n_y + cy[l])%n_y
```

```
55              iZ = (k + n_z + cz[l])%n_z
56              Fin[iX][iY][iZ][l] = Fout[i][j][k][l]
57            end
58          end
59        end
60    end
```

This code leads to a single node performance of 4.13 Million Lattice Updates per Second (MLUPS) with the breakdown of the time spend in different parts of the code shown in table 1. Even though the streaming part of the algorithm involves no floating point operations it takes the longest. This is due to logic for the periodic domain employed in this timing example. The algorithm has much scope for improvement. For example, loops can be merged together and the inner loop can be unrolled. One might hope that much of this work could be done within the optimization of the compiler but, as will be shown, this is not the case.

Calculation	Time	Percentage time
Continuum	19.79s	20.66%
Collision	33.21s	34.66%
Streaming	42.80s	44.68%
Total	95.80s	100.%

Table 1 – CPU time for the three stages of the LBM algorithm 1. Times obtained on a Intel Xeon E3-1245 CPU running at 3.30GHz

3.1.1. Ordering the indices

The basic implementation of Listing 1 is independent on the ordering of the $\bar{e}_i$. However, the ordering might effect the performance of the streaming step, as memory access operations for the `Fin` array depend on it. Four different orderings where tested as outlined in figure 4. The first three are common D3Q19 orderings which all have the resting particles at index 0, however, the last ordering steps sequentially through the memory so the resting particles are in the middle, at index 9. Table 2 shows the timings for Listing 1 with the four different orderings. There was a small, three percent difference between the fastest and the slowest streaming with the different orderings, though other parts of the algorithm were mainly unchanged. The only ordering which had an effect on the rest of the algorithm was the final one (Order 4 of table 2 and Figure 4) which had approximately 5 percent improvement in the calculation of the continuum variables. Therefore, only ordering 4 will be used from now on.

Ordering	Continuum	Collision	Streaming	Total
Order 1	20.11s	33.78s	42.70s	96.59s
Order 2	20.16s	33.77s	43.41s	97.34s
Order 3	20.07s	33.82s	42.92s	96.81s
Order 4	18.81s	33.45s	41.96s	94.22s

Table 2 – CPU time for the three stages of the LBM Listing 1 with four different orderings.

3.1.2. Unrolling the streaming inner loop

The first loop to be unrolled was the streaming loop. The new streamed indices for each lattice point use 57 modulus where there are only 3 unique pairs $i \pm 1$, $j \pm 1$ and $k \pm 1$ for each lattice point. In fact, even fewer are required as $i \pm 1$ are the same for the whole j-k plane so the total number of updates for the lattice points is

$$2(n_x + n_x \times n_y + n_x \times n_y \times n_z) \tag{11}$$

which is a little over 2 per lattice point. The changes to the streaming step for indexing method 4 are as seen in Listing 2.

Listing 2 – Unrolled streaming loop

```
1   /* 4th Loop Streaming Step */
2   for i = 0 to nx − 1 do
3     rig = (i+1)%nx
4     lef = (i+nx−1)%nx
5     for j = 0 to ny − 1 do
6       top = (j+1)%ny
7       bot = (j+ny−1)%ny
8       for k = 0 to nz − 1 do
9         bac = (k+1)%nz
10        fro = (k+nz−1)%nz
11        Fin[lef][bot][  k][0] = Fout[i][j][k][0]
12        Fin[lef][  j][fro][1] = Fout[i][j][k][1]
13        Fin[lef][  j][  k][2] = Fout[i][j][k][2]
14        Fin[lef][  j][bac][3] = Fout[i][j][k][3]
```

```
15            Fin[lef][top][  k][4] = Fout[i][j][k][4]
16            Fin[  i][bot][fro][5] = Fout[i][j][k][5]
17            Fin[  i][bot][  k][6] = Fout[i][j][k][6]
18            Fin[  i][bot][bac][7] = Fout[i][j][k][7]
19            Fin[  i][  j][fro][8] = Fout[i][j][k][8]
20            Fin[  i][  j][  k][9] = Fout[i][j][k][9]
21            Fin[  i][  j][bac][10] = Fout[i][j][k][10]
22            Fin[  i][top][fro][11] = Fout[i][j][k][11]
23            Fin[  i][top][  k][12] = Fout[i][j][k][12]
24            Fin[  i][top][bac][13] = Fout[i][j][k][13]
25            Fin[rig][bot][  k][14] = Fout[i][j][k][14]
26            Fin[rig][  j][fro][15] = Fout[i][j][k][15]
27            Fin[rig][  j][  k][16] = Fout[i][j][k][16]
28            Fin[rig][  j][bac][17] = Fout[i][j][k][17]
29            Fin[rig][top][  k][18] = Fout[i][j][k][18]
30          end
31        end
32      end
```

This loop is now not independent of the numbering within the lattice. The timings for two numberings are shown in table 3. The unrolled streaming loop is around two and a half times faster than the original loop and the performance increase from using lattice ordering 4 is maintained. There is no performance gain from moving the indices `lef`, `rig`, `top` and `bot` (see e.g. lines 3 and 4 of Listing 2 to the loops where they are incremented since the compiler optimizer also did this.

Ordering	Continuum	Collision	Streaming	Total
Order 2	19.83s (28.62%)	33.15s (47.84%)	16.31s (23.54%)	69.29s
Order 4	18.93s (27.81%)	33.68s (49.49%)	15.44s (22.70%)	68.05s

Table 3 – CPU time for the three stages of the LBM algorithm with unrolled streaming loop for 2 different orderings.

3.1.3. Unrolling and merging the continuum loops

Next, the continuum loops were unrolled and merged together. The unrolling of the density loop `rho` showed very small performance increase, however, unrolling the velocity loop had a larger performance gain. This was even the case when `ex`, `ey` and `ez` where defined as constant, global, arrays. The final unrolled loops are shown in Listing 3.

Listing 3 – Improved continuum loops

```
1  /* First and Second Loops Calculate values of rho,u,v,w */
2  for i = 0 to n_x - 1 do
3    for j = 0 to n_y - 1 do
4      for k = 0 to n_z - 1 do
5        fTemp=Fin[i][j][k][ 0]+Fin[i][j][k][ 1]+Fin[i][j][k][ 2]
6              +Fin[i][j][k][ 3]+Fin[i][j][k][ 4]+Fin[i][j][k][ 5]
7              +Fin[i][j][k][ 6]+Fin[i][j][k][ 7]+Fin[i][j][k][ 8]
8              +Fin[i][j][k][ 9]+Fin[i][j][k][10]+Fin[i][j][k][11]
9              +Fin[i][j][k][12]+Fin[i][j][k][13]+Fin[i][j][k][14]
10             +Fin[i][j][k][15]+Fin[i][j][k][16]+Fin[i][j][k][17]
11             +Fin[i][j][k][18]
```

```
12          rho[i][j][k] = fTemp
13          Tmpu=-Fin[i][j][k][ 0]-Fin[i][j][k][ 1]-Fin[i][j][k][ 2]
14                -Fin[i][j][k][ 3]-Fin[i][j][k][ 4]+Fin[i][j][k][14]
15                +Fin[i][j][k][15]+Fin[i][j][k][16]+Fin[i][j][k][17]
16                +Fin[i][j][k][18]
17          Tmpv=-Fin[i][j][k][ 0]+Fin[i][j][k][ 4]-Fin[i][j][k][ 5]
18                -Fin[i][j][k][ 6]-Fin[i][j][k][ 7]+Fin[i][j][k][11]
19                +Fin[i][j][k][12]+Fin[i][j][k][13]-Fin[i][j][k][14]
20                +Fin[i][j][k][18]
21          Tmpw=-Fin[i][j][k][ 1]+Fin[i][j][k][ 3]-Fin[i][j][k][ 5]
22                +Fin[i][j][k][ 7]-Fin[i][j][k][ 8]+Fin[i][j][k][10]
23                -Fin[i][j][k][11]+Fin[i][j][k][13]-Fin[i][j][k][15]
24                +Fin[i][j][k][17]
25
26          ux[i][j][k] = Tmpu/rho[i][j][k]
27          uy[i][j][k] = Tmpv/rho[i][j][k]
28          uz[i][j][k] = Tmpw/rho[i][j][k]
29        end
30      end
31  end
```

This leads to the performance shown in table 4. There was a good improvement in the continuum calculation time and most of this was due to merging the two loops over the distribution functions so memory had to be traversed only once.

Calculation	Time	Percentage time
Continuum	7.64s	13.56%
Collision	33.34s	59.13%
Streaming	15.39s	27.31%
Total	56.37s	100.0%

Table 4 – CPU time for the three stages of the LBM Listing 3 with unrolled continuum and streaming loops.

3.1.4. Unrolling the collision loop

The collision step now takes about 60% of the runtime of the code. The same loop unrolling was applied here, and a number of temporary variables were added to reduce the overall number of floating point operations. Listing 4 presents the necessary changes.

Listing 4 – Improved collision loop

```
1  /* 3rd Loop Collision Step */
2  for i = 0 to n_x - 1 do
3    for j = 0 to n_y - 1 do
4      for k = 0 to n_z - 1 do
5        tux = 3.0 * ux[i][j][k]
6        tuy = 3.0 * uy[i][j][k]
7        tuz = 3.0 * uz[i][j][k]
8        trho = rho[i][j][k]
9
10       uxyz2 =1.0 - 0.166666666667*(tux*tux + tuy*tuy + tuz*tuz)
```

```
11          ux2 = 0.5 * tux*tux
12          uy2 = 0.5 * tuy*tuy
13          uz2 = 0.5 * tuz*tuz
14          uxy2 = ux2+uy2
15          uxz2 = ux2+uz2
16          uyz2 = uy2+uz2
17          uxy = tux*tuy
18          uxz = tux*tuz
19          uyz = tuy*tuz
20
21          Fout[i][j][k][ 0]=Fin[i][j][k][ 0] - omega*(Fin[i][j][k][ 0]-
22                           (w[ 0]*trho*(uxyz2 - tux - tuy + uxy2 + uxy)))
23          Fout[i][j][k][ 1]=Fin[i][j][k][ 1] - omega*(Fin[i][j][k][ 1]-
24                           (w[ 1]*trho*(uxyz2 - tux - tuz + uxz2 + uxz)))
25          Fout[i][j][k][ 2]=Fin[i][j][k][ 2] - omega*(Fin[i][j][k][ 2]-
26                           (w[ 2]*trho*(uxyz2 - tux + ux2 )))
27          Fout[i][j][k][ 3]=Fin[i][j][k][ 3] - omega*(Fin[i][j][k][ 3]-
28                           (w[ 3]*trho*(uxyz2 - tux + tuz + uxz2 - uxz)))
29          Fout[i][j][k][ 4]=Fin[i][j][k][ 4] - omega*(Fin[i][j][k][ 4]-
30                           (w[ 4]*trho*(uxyz2 - tux + tuy + uxy2 - uxy)))
31          Fout[i][j][k][ 5]=Fin[i][j][k][ 5] - omega*(Fin[i][j][k][ 5]-
32                           (w[ 5]*trho*(uxyz2 - tuy - tuz + uyz2 + uyz)))
33          Fout[i][j][k][ 6]=Fin[i][j][k][ 6] - omega*(Fin[i][j][k][ 6]-
34                           (w[ 6]*trho*(uxyz2 - tuy + uy2 )))
35          Fout[i][j][k][ 7]=Fin[i][j][k][ 7] - omega*(Fin[i][j][k][ 7]-
```

```
36                          (w[ 7]*trho*(uxyz2 - tuy + tuz + uyz2 - uyz)))
37          Fout[i][j][k][ 8]=Fin[i][j][k][ 8] - omega*(Fin[i][j][k][ 8]-
38                          (w[ 8]*trho*(uxyz2 - tuz + uz2 )))
39          Fout[i][j][k][ 9]=Fin[i][j][k][ 9] - omega*(Fin[i][j][k][ 9]-
40                          (w[ 9]*trho*(uxyz2)))
41          Fout[i][j][k][10]=Fin[i][j][k][10] - omega*(Fin[i][j][k][10]-
42                          (w[10]*trho*(uxyz2 + tuz + uz2 )))
43          Fout[i][j][k][11]=Fin[i][j][k][11] - omega*(Fin[i][j][k][11]-
44                          (w[11]*trho*(uxyz2 + tuy - tuz + uyz2 - uyz)))
45          Fout[i][j][k][12]=Fin[i][j][k][12] - omega*(Fin[i][j][k][12]-
46                          (w[12]*trho*(uxyz2 + tuy + uy2 )))
47          Fout[i][j][k][13]=Fin[i][j][k][13] - omega*(Fin[i][j][k][13]-
48                          (w[13]*trho*(uxyz2 + tuy + tuz + uyz2 + uyz)))
49          Fout[i][j][k][14]=Fin[i][j][k][14] - omega*(Fin[i][j][k][14]-
50                          (w[14]*trho*(uxyz2 + tux - tuy + uxy2 - uxy)))
51          Fout[i][j][k][15]=Fin[i][j][k][15] - omega*(Fin[i][j][k][15]-
52                          (w[15]*trho*(uxyz2 + tux - tuz + uxz2 - uxz)))
53          Fout[i][j][k][16]=Fin[i][j][k][16] - omega*(Fin[i][j][k][16]-
54                          (w[16]*trho*(uxyz2 + tux + ux2 )))
55          Fout[i][j][k][17]=Fin[i][j][k][17] - omega*(Fin[i][j][k][17]-
56                          (w[17]*trho*(uxyz2 + tux + tuz + uxz2 + uxz)))
57          Fout[i][j][k][18]=Fin[i][j][k][18] - omega*(Fin[i][j][k][18]-
58                          (w[18]*trho*(uxyz2 + tux + tuy + uxy2 + uxy)))
59        end
60      end
```

61 **end**

The temporary variables were set up so that the constants in the equilibrium distribution are already taken into account. Each lattice direction has been reduced from 8 additions and 10 multiplications, to at most 6 additions and 3 multiplications. This leads to the timings shown in table 5 suggesting a reduction in the collision step of over 50%.

Calculation	Time	Percentage time
Continuum	7.66s	19.96%
Collision	15.37s	40.01%
Streaming	15.38s	40.03%
Total	38.41s	100.0%

Table 5 – CPU time for the three stages of the LBM algorithm with unrolled continuum and streaming loops.

3.1.5. Removing all temporary storage by merging all loops

The last optimization is to merge all steps into one loop removing all other temporary storage, at the cost of having to recalculate the continuum variables when output is required. However, this produced some very interesting timing results. First the Continuum and Collision loops where merged as shown in Listing 5.

Listing 5 – Removed temporary storage

```
1  /* Merging 1st 2nd and 3rd Loops from basic implementation */
2  for i = 0 to n_x − 1 do
```

```
3       for j = 0 to n_y − 1 do
4         for k = 0 to n_z − 1 do
5           trho=Fin[i][j][k][ 0]+Fin[i][j][k][ 1]+Fin[i][j][k][ 2]
6                +Fin[i][j][k][ 3]+Fin[i][j][k][ 4]+Fin[i][j][k][ 5]
7                +Fin[i][j][k][ 6]+Fin[i][j][k][ 7]+Fin[i][j][k][ 8]
8                +Fin[i][j][k][ 9]+Fin[i][j][k][10]+Fin[i][j][k][11]
9                +Fin[i][j][k][12]+Fin[i][j][k][13]+Fin[i][j][k][14]
10               +Fin[i][j][k][15]+Fin[i][j][k][16]+Fin[i][j][k][17]
11               +Fin[i][j][k][18]
12          Tmpu=−Fin[i][j][k][ 0]−Fin[i][j][k][ 1]−Fin[i][j][k][ 2]
13               −Fin[i][j][k][ 3]−Fin[i][j][k][ 4]+Fin[i][j][k][14]
14               +Fin[i][j][k][15]+Fin[i][j][k][16]+Fin[i][j][k][17]
15               +Fin[i][j][k][18]
16          Tmpv=−Fin[i][j][k][ 0]+Fin[i][j][k][ 4]−Fin[i][j][k][ 5]
17               −Fin[i][j][k][ 6]−Fin[i][j][k][ 7]+Fin[i][j][k][11]
18               +Fin[i][j][k][12]+Fin[i][j][k][13]−Fin[i][j][k][14]
19               +Fin[i][j][k][18]
20          Tmpw=−Fin[i][j][k][ 1]+Fin[i][j][k][ 3]−Fin[i][j][k][ 5]
21               +Fin[i][j][k][ 7]−Fin[i][j][k][ 8]+Fin[i][j][k][10]
22               −Fin[i][j][k][11]+Fin[i][j][k][13]−Fin[i][j][k][15]
23               +Fin[i][j][k][17]
24
25          tux = Tmpu/trho
26          tuy = Tmpv/trho
27          tuz = Tmpw/trho
```

```
28
29            uxyz2 =1.0 - 0.166666666667*(tux*tux + tuy*tuy + tuz*tuz)
30            ux2 = 0.5 * tux*tux
31            uy2 = 0.5 * tuy*tuy
32            uz2 = 0.5 * tuz*tuz
33            .
34            .
35            .
36          end
37        end
38      end
```

This leads to a combined time of 25.69 seconds which was in fact slower than the previous version of the code. It is thought that this is probably due to a pipeline stall while calculating `tux`, `tuy` and `tuz` as they are required to be calculated before the rest of the loop can be executed. This was confirmed with a very small change to the algorithm. After `trho` is calculated then `rrho=1.0/trho` was calculated, and used in the calculations of `tux`, `tuy` and `tuz`. This reduced the combined time down to 22.98 seconds which is now the same time as using the uncombined loops while removing the extra storage needed for the continuum variables. However, it should still be possible to do better than this as the loop overhead has been removed from the code. This was achieved by the following algorithm (Listing 6) in which the continuum variables are stored for each k line.

Listing 6 – Improving the pipeline stall

```
1   /* Calculate the macroscopic values of rho,u,v,w */
2   for i = 0 to n_x - 1 do
3     for j = 0 to n_y - 1 do
4       /* Calculate continuum Variables for [i][j][] */
5       for k = 0 to n_z - 1 do
6         trho=...
7         rrho=1.0/trho
8         Tmpu=...
9         Tmpv=...
10        Tmpw=...
11
12        ttrho[k] = trho
13        ttux[k] = Tmpu*rrho
14        ttuy[k] = Tmpv*rrho
15        ttuz[k] = Tmpw*rrho
16      end
17      /* Collision step for [i][j][] */
18      for k = 0 to n_z - 1 do
19
20        tux = 3.0 * ttux[k];
21        tuy = 3.0 * ttuy[k];
22        tuz = 3.0 * ttuz[k];
23        trho = ttrho[k];
24
25        uxyz2 =1.0 - 0.166666666667*(tux*tux + tuy*tuy + tuz*tuz)
```

```
26            ux2 = 0.5 * tux*tux
27            uy2 = 0.5 * tuy*tuy
28            uz2 = 0.5 * tuz*tuz
29            .
30            .
31            .
32          end
33        end
34      end
```

This reduced the computational cost down to 19.53 seconds for the test block which is a good saving. Finally, the streaming step was added into the collision loop and the total run time was increased by only 2.98 seconds for a total of 22.81. This means that the total savings for full merging the streaming with the collision steps, and part merging the continuum step, reduced the runtime of the code from 38.41s to 22.81s or by about 40%.

As a final remark, the overall reduction from the basic algorithm to this one was from 95.8 seconds to 22.81 seconds, meaning that the performance has increased from 4.17MLUPS to 17.53MLUPS which is over 4 times faster.

3.1.6. Effect of block size on different algorithms

We now have two different LBM implementations. Algorithm 1 is Listing 5 with the precalculation of `rrho=1.0/trho` and the second algorithm is Listing 6 which has two loops over `k`: one for the continuum, and one for the collision and streaming. The second algorithm looks like it might be more depended on block size as the number of continuum variables stored

increased. Both algorithms were tested using square lattice blocks of increasing size, and a fixed number of lattice points. As can be seen from tables 6 and 7, there is a drop in performance as the block size is increased which can be delayed by using floating point precision instead of doubles. It can also be seen that the second algorithm also becomes slower than the first when the block size reaches 32. The first algorithm performs better and has less performance drop-off at large block sizes.

Block Size	Algorithm 1	Algorithm 2	Algorithm 2 Floats
$16 \times 16 \times 16$	17.94	19.93	20.71
$24 \times 24 \times 24$	16.96	18.77	20.13
$32 \times 32 \times 32$	16.16	16.12	20.47
$64 \times 64 \times 64$	16.50	14.92	17.85
$96 \times 96 \times 96$	15.62	14.40	17.40
$128 \times 128 \times 128$	14.81	13.36	17.19

Table 6 – Performance of both algorithms in MLUPS for square lattices.

3.1.7. Swapping instead of temporary data

In Listing 1, the variables `Fin` and `Fout` cannot be identical because values that are still needed get overwritten. The swapping algorithm introduced by Latt (2007) has `Fin` and `Fout` sharing the same data space but uses different indices for the population functions. An opposite `opp` is defined for $l \in [0, 1, \ldots, q-1]$ with respect to the following relation

$$c_{\mathtt{opp}(l)} = -c_l. \tag{12}$$

Block Size	Algorithm 1	Algorithm 2
$128 \times 128 \times 128$	14.81	13.36
$256 \times 256 \times 32$	16.42	15.84
$512 \times 512 \times 8$	16.52	17.70
$1024 \times 512 \times 4$	16.40	17.47
$512 \times 256 \times 16$	16.35	17.43

Table 7 – Performance of both algorithms in MLUPS for a fixed number of lattice points.

Then, only a single set of distribution functions `F` are used with the incoming functions stored at the normal location $\texttt{Fin}[l] = \texttt{F}[l]$ and the outgoing functions $\texttt{Fout}[l] = \texttt{F}[\texttt{opp}(l)]$ which happens on line 12 and lines 15,30 respectively. The indices are then reordered so that particles at equilibrium have index 0 and the opposite of the index in the lower half is found by adding $(q-1)/2$, as shown in figure 1. Hence

$$\texttt{opp}(l) = l + (q-1)/2 \quad \text{for} \quad l = 1, 2, 3, \ldots, (q-1)/2 \tag{13}$$

Listing 7 presents the code changes for the collision and streaming steps.

Listing 7 – Basic swap implementation

```
1   /* Collision Step with Swaps */
2   half = (q-1)/2
3   for i=0 to n_x - 1 do
4     for j=0 to n_y - 1 do
5       for k=0 to n_z - 1 do
6         v3 = ux[i][j][k] * ux[i][j][k] + uy[i][j][k] * uy[i][j][k]
```

```
7                 + uz[i][j][k] * uz[i][j][k]
8             for l = 0 to q-1 do
9               v1=ex[l]*ux[i][j][k]+ey[l]*uy[i][j][k]+ez[l]*uz[i][j][k]
10              v2=vel1 * vel1
11              Feq  = rho[i][j][k]*w[l]*(1.0+3.0*v1+4.5*v2-1.5*v3)
12              F[i][j][k][l] = (1.0-omega)*F[i][j][k][l]+omega*Feq
13            end
14            for l=1 to half do
15              swap(F[i][j][k][l],F[i][j][k][l+half])
16            end
17        end
18      end
19    end
20
21    /* Streaming Step by Swapping */
22    for i=0 to n_x - 1 do
23      for j=0 to n_y - 1 do
24        for k=0 to n_z - 1 do
25          for l = 1 to half do
26            iX = i + cx[l]
27            iY = j + cy[l]
28            iZ = k + cz[l]
29            if (iX>=0 && iX < n_x && iY>=0 && iY < n_y iZ>=0 && iZ < n_z)
30              swap(F[i][j][k][l+half],F[iX][iY][iZ][l])
31            endif
```

```
32            end
33          end
34        end
35      end
```

There are two additional properties with the above algorithm. The distribution functions $\mathtt{F}_l$ which would stream out of the domain stay in place at $\mathtt{F}_{\mathtt{opp}(l)}$. This is possible because this location corresponds to the distribution functions streaming from outside the domain and hence does not exist. The second is that the memory can be stepped through in any order. This is important for graphics processing units (GPUs) where there is less control over the order in which operations occur.

3.1.8. Traversing the Memory only once

When swaps are used, there is a specific way to step through the memory only once. This is done by only swapping the distribution functions which have already been computed, at the next time level. This can be achieved by reordering the indices as described in ordering 1 and figure 4. Listing 8 shows the new code.

Listing 8 – Collide and stream implementation for single memory pass

```
1   /* Collision and Stream with Swaps */
2   half = (q-1)/2
3   for i=0 to n_x - 1 do
4     for j=0 to n_y - 1 do
5       for k=0 to n_z - 1 do
6         v3 = ux[i][j][k] * ux[i][j][k] + uy[i][j][k] * uy[i][j][k]
```

```
7                 + uz[i][j][k] * uz[i][j][k]
8             for l = 0 to q-1 do
9               v1=ex[l]*ux[i][j][k]+ey[l]*uy[i][j][k]+ez[l]*uz[i][j][k]
10              v2=vel1 * vel1
11              Feq  = rho[i][j][k]*w[l]*(1.0+3.0*v1+4.5*v2-1.5*v3)
12              F[i][j][k][l] = (1.0-omega)*F[i][j][k][l]+omega*Feq
13            end
14            for l=1 to half do
15              iX = i + cx[l]
16              iY = j + cy[l]
17              iZ = k + cz[l]
18              if (iX>=0 && iX < n_x && iY>=0 && iY < n_y iZ>=0 && iZ < n_z)
19                fTmp = F[i][j][k][l]
20                F[i][j][k][l] = F[i][j][k][l+half]
21                F[i][j][k][l+half] = F[iX][iY][iZ][l]
22                F[iX][iY][iZ][l] = fTmp
23              endif
24            end
25          end
26        end
27    end
```

This is the algorithm that both Palabos (2016) and Openlb (2016) use and it does suffer from not having the ability to step through the memory space in any order but the memory is accessed only once instead of twice.

3.2. Implementation of the Entropic lattice Boltzmann method (ELBM)

Although equation 8 is local it is much more expensive to calculate than equation 5, i.e. the usual LBM equilibrium function. It should be noted that the exponents take the values of zero and ± 1 only and hence the expressions for the equilibrium function do not need power functions. The performance of five different loops where tested. The first is a simple loop shown in Listing 9. Where the compiler knows that the `cx`, `cy` and `cz` are constants and, that only take values of 0 and ± 1. The addition of this simple loop is about 13 times slower than the optimized BGK Lattice Boltzmann method of algorithm 2.

Listing 9 – Basic Implementation ELMB equilibrium function

```
1   for i = 0 to n_x - 1 do
2     for j = 0 to n_y - 1 do
3       /* Calculate continuum Variables for [i][j][] */
4       for k = 0 to n_z - 1 do
5         .
6         .
7       end
8       /* Collision step for [i][j][] */
9       for k = 0 to n_z - 1 do
10        tux = ttux[k];
11        tuy = ttuy[k];
12        tuz = ttuz[k];
13        trho = ttrho[k];
14        for l = 0 to q - 1 do
```

```
15              prod = 1.0;
16              prod = prod*(2-sqrt(1+3*tux*tux))*
17                     pow((2*tux+sqrt(1+3*tux*tux))/(1-tux),cx[l])
18              prod = prod*(2-sqrt(1+3*tuy*tuy))*
19                     pow((2*tuy+sqrt(1+3*tuy*tuy))/(1-tuy),cy[l])
20              prod = prod*(2-sqrt(1+3*tuz*tuz))*
21                     pow((2*tuz+sqrt(1+3*tuz*tuz))/(1-tuz),cz[l])
22              Ftmp[l] = Fin[i][j][k][l]
23                      - omega*(Fin[i][j][k][l]-w[l]* trho * prod)
24            end
25          end
26        end
27      end
```

The first optimization was to pull the constant term

$$\sum_{\alpha=1}^{3}\left(2-\sqrt{1+3u_{\alpha}^{2}}\right) \tag{14}$$

outside of the inner loop. This reduced the computational cost by about 1/3 and shows that the compiler failed to make this optimization. Unrolling the inner loop at line 14 showed a marked improvement in speed with a doubling of the performance. With the inner loop unrolled, the power functions originally on lines 17,19 and 21 are now repeated q times but the second argument is now fixed and so all the functions can be replaced by either 1, the first argument, and the reciprocal of the first argument. This resulted in a marked increase in performance with the code running at around 5 million lattice updates per seconds. It should be noted that only replacing the pow-

ers of -1 with the reciprocal leaded to improvements so, the standard power function must test of cases when the second argument is 0 or 1. The final optimization used temporary variables store the first arguments of the `pow` function on lines 17,19 and 21 and their reciprocals leads to a performance of over 11 MLUPS. This is 50% slower than the BGK version of the code, due to three extra square roots, and three divisions per lattice point. Also, these operations will not allow the CPU pipeline to be full utilized since the unrolled inner loop is depends on them.

Method	MLUPS	Slowdown
Simple loop	1.32	13.16
Precalculated Constant	1.84	9.44
Unrolled Loop	3.86	4.50
Unrolled loop no power function	5.25	3.31
Unrolled loop plus temp variables	11.73	1.51

Table 8 – CPU time for the three stages of the LBM algorithm with unrolled continuum and streaming loops.

3.2.1. Solution of the h-function monotonicity constraint

The solution of the h-function monotonicity constraint, equation 9, is very computationally expensive. The function has 19 natural logarithms in each function evaluation. Although 19 evaluations of ln is of the same order as the number of added divisions and square roots, the number of cycles to evaluate a natural logarithm is twenty times longer than either a division or a square root. This means that the algorithm will quickly become impossible to run

in real time if many h-function evaluations are needed on today's hardware. However, if the solution is close to the equilibrium function then $\beta = 2$ and hence Newton's method would not have to be used, saving much CPU time. It is possible to expand the natural logarithm as follows

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \ldots \qquad -1 < x \leq 1. \tag{15}$$

where

$$x = \sum_i \frac{f_i^{eq} - f_i}{f_i}. \tag{16}$$

Even this benign test of calculating x requires 19 divisions and reduces the performance of the code from 11.0MLUPS to 4.60MLUPS.

3.2.2. Newton's Method

Newton's method can be used to solve

$$H(f) = H(f + \beta(f^{eq} - f)) \qquad \text{where,} \qquad H(f) = \sum_{i=1}^{q} f_i \ln\left(\frac{f_i}{w_i}\right). \tag{17}$$

One step of Newton's iteration for this system is

$$\beta^{n+1} = \beta^n - \frac{G(f,\beta)}{\partial G(f,\beta)/\partial\beta} \tag{18}$$

where

$$G(f,\beta) = \sum_{i=1}^{q} (f_i + \beta(f_i^{eq} - f_i)) \ln\left(\frac{f_i + \beta(f^{eq} - f)}{w_i}\right) - f_i \ln\left(\frac{f_i}{w_i}\right). \tag{19}$$

and

$$\frac{\partial G(f,\beta)}{\partial\beta} = \sum_{i=1}^{q} (f_i^{eq} - f_i) \left[1.0 + \ln\left(\frac{f_i + \beta(f^{eq} - f)}{w_i}\right)\right]. \tag{20}$$

There are a total of $3q$ natural logarithms in equations 19 and 20 but two can be removed because one is independent of β and the other two are the same but scaled differently as shown in Listing 10.

Listing 10 – Optimized Newton's method for solving the h-function

```
1   Ftmp[l] = feq[i][j][k][l] - Fin[i][l][k][l]
2   /* Solution of Beta by Newtons method */
3   entc = 0.0
4   for l = 0 to q - 1 do
5     entc += Fin[i][j][k][l] * log(Fin[i][j][k][l]/w[l])
6   end
7   for Nstep = 0 to Nits do
8     /* DO Newton step */
9     for l = 0 to q - 1 do
10      FNeq = fin[i][j][k][l] + beta * Ftmp[l]
11      tmpl = log(FNeq/w[l])
12      tmp1 += FNeq * tmpl
13      tmp2 += Ftmp[l] * (1.0 + tmpl)
14    end
15    betan = beta - (tmp1 - entc)/tmp2;
16    error = fabs(betan-beta)
17    beta = betan
18  end
```

The calculation of a single h-function adds a factor of 16 to the overall computational cost of a lattice point. So, just two Newton iterations which have a total of 3 h-function evaluations increase the cost by a factor of 50. This can be reduced by about 4% if the division in the logarithm is removed and replace by the correct pre-calculated constant value. It should be noted that a check has to be made for the case when `Ftmp` is very small as rounding

error can lead to the solution moving away from the solution $\beta = 2.0$.

3.2.3. Approximate solutions

Another method needs to be found to reduce the computational effort. This was done by using an approximate solution introduced by Chikatamarla et al. (2006) for when `Ftmp` is small with respect to `Fin`. This avoids using Newton's method altogether. Chikatamarla suggests the following approximate solution.

$$\beta = 2 - \frac{4b_2}{b_1} + \frac{16b_2^2}{b_1^2} - \frac{8b_3}{b_1} + \frac{80b_2b_3}{b_1^2} - \frac{80b_2^3}{b_1^3} - \frac{16b_4}{b_1} \tag{21}$$

where

$$b_n = \frac{-1^{n-1}}{n(n+1)} \sum_i \frac{(f_i^{eq} - f_i)^{(n+1)}}{f_i^n}. \tag{22}$$

This leads to the performance given in table 9. So, even doing just one

Baseline	Approximate	1 Newton Step	2 Newton Steps
17.42MLUPS	4.17MLUPS	0.34MLUPS	0.21MLUPS

Table 9 – Performance in lattice updates per second for different ways of solving the h-function in ELBM.

Newton step for all lattice points reduces the performance by nearly a factor of 20. Even if just 10% need a single Newton step the ELBM will still be a factor of 5 times slower. The 4.17MLUPS for the approximate solution is similar to the performance of just including the test function. This is because when test function is calculated very little extra work is required as the expensive part $(f_i^{eq} - f_i)/f_i$ has already been stored, and does not need recalculating.

3.3. Unequal lattice cost

The ELMB algorithm has three different paths depending on the magnitude of equation 16 and each of these paths has a different computational cost associated with it. Firstly, there is the case of equation 16 being very small in which case β is set to 2. Secondly, there is the case where approximation (21) is valid, and lastly the case where Newton's method is required to calculate β. This means that a partition of the domain into equal-sized blocks will not, in general, give equal work load per processor. Even if the domain is partitioned in such a way that the work load is initially balanced, as the flow evolves the method of calculating β will change resulting in a changing work load, and the need for periodic re-balancing.

4. Results and Discussion

The following section first presents the LBM parallel performance on a $121 \times 241 \times 241$ lattice containing 7 million lattice points with periodic boundary conditions in all three directions. Next, two dimensional flows around both a cylinder and the three dimensional flow around the full SFS2 will be presented with a discussion of the computing power required for real-time performance.

4.1. LBM Solver Performance on CPUs

The test problem used to examine the parallel performance is a 7 million point lattice with periodic boundary conditions in all three directions. The lattice used a Cartesian partition of $NX_p \times NY_p \times NZ_p$ equal sized blocks. The total number of blocks equalled the total number of processors to maximize

parallel performance. The original LBM of the University of Glasgow did not have any of the efficiency gains what are implemented within Palabos (2016) such as the swap approach in the collision and streaming steps, combining the collision and streaming steps to only have one loop over the memory, and improved memory layouts [Wellein et al. (2006); Mattila et al. (2008)]. Hence there was a large scope to increase the overall single-node performance of the code which was shown in section 3. After these optimizations the single-node performance exceeded both Palabos (2016) and Openlb (2016).

The parallel performance of the current LBM code was compared to Palabos within a compute node of ARCHER[1] with the results shown in figure 5(a). They both have very similar curves and show a marked drop off in parallel performance when running on more that 4 core per node - 8 codes in total. This is because the method is very memory bandwidth intensive and general memory bandwidth has not kept pace with the ever increasing number of cores on CPUs. However, the performance across nodes shows linear speedup going from 1 node (24 cores) to 64 nodes (1536 cores). This is because the number of lattice points per process dropped from $288,000$ when on a single node to just 4500 when on 64. This means a much larger percentage of the data could be stored in the cache which increases the core performance, as shown in table 6. This twenty percent gain in sequential performance offsets the communication costs.

[1]ARCHER (Advanced Research Computing High End Resource) is the UK National Supercomputing Service.

4.2. LBM Solver Performance on Knights Landing Processors

Another system used here employs knights landing (KNL) processors which are second generation Intel Xeon Phi Many Integrated Core (MIC) processors. Each of the computer nodes contains a 64-core KNL processors (model 7210) running at 1.3Ghz. These offer a large amount of floating point performance, (3TFlops peak using double precision) and the hardware is a significant step forward from the previous generation of Xeon Phis.

The code was evaluated on nodes configured in cache mode with all 16GB of the on-chip Multi-Channel DRAM (MCDRAM) used to cache the system memory, and job sizes were small enough so all the data could fit within the cache. The MCDRAM is a high bandwidth memory which fits well with the needs of a LBM. The results can be seen in figure 6 and although the single core performance of a KNL processor was three times slower, mainly due to the lower clock speed, the parallel scaling was much better at high number of cores. Table 10 shows that even with better scaling the KNL processor is still slower when comparing at the same number of cores. However, due to the KNL having additional cores when comparing fully utilized nodes, it is 80% faster. A more detailed comparison, including multiple node performance, can be found in the KNL Performance Comparison reportBarakos and Woodgate (2017).

Cores	Xeon		KNL	
Cores	Time	Efficiency	Time	Efficiency
1	2.1249s	N/A	6.772s	N/A
2	1.0721s	99.1%	3.503s	96.6%
4	0.56952s	93.2%	1.743s	97.1%
8	0.34484s	77.0%	0.880s	96.2%
16	0.23538s	56.4%	0.442s	95.6%
24	0.21953s	41.3%	N/A	N/A
32	N/A	N/A	0.226s	93.6%
64	N/A	N/A	0.126s	84.0%

Table 10 – Performance comparison between Xeon and KNL systems.

It should be noted that both Listings 5 and 6 have their inner loops unrolled and hence make no use of the 2 vector processing units (VPU) per core. A listing which takes full advantage of the VPUs might increase the single core performance and is currently under investigation.

4.3. Co-rotating Vortex Pair

The first test case is a pair of two dimensional co-rotating equal vortices. Before merging, the vortex pair can be described, as follows. Firstly, the two vortices rotate around the centroid of the total vortex. Next the vorticity contours are deformed elliptically by the local strain induced by each vortex on the other. Thirdly, the vorticity waves propagate from each vortex rearranging the field and lastly, the core of each vortex increases due to viscous effects. The viscosity reduces the ratio between vortex radius and vortex separation until a critical value is reached and the vortices merge. Figure 7

shows the merging of a co-rotating vortex pair on a 801×801 lattice with lattice viscosity of 0.01 at every 2000 time steps. Due to the low lattice viscosity the vortices merge in just under one revolution.

4.4. Counter-rotating Vortex Pair

For a counter-rotating vortex pair in a viscous fluid, their mutually-induced velocities cause them to remain at a fixed distance apart, and to move together in a fixed direction. Figure 8 shows the initial solution on the baseline 101×201 lattice, and the fine 201×401 with a lattice viscosity of 0.01. The peak magnitude of vorticity is the same for both lattices, while the behaviour near the peak is less well represented on the the baseline lattice. Figure 9 shows the comparison between the vorticity on the baseline and fine lattice computations after cycling through the domain 4 times which takes 21 thousand time-steps on the baseline lattice and 42 thousand time-steps for the fine lattice computation. The solutions are very similar with the baseline lattice only slightly lagging behind the finer grid solution with also a slightly reduced peak vorticity. This means that the resolution of 101×201 of the baseline lattice is a lattice-converged solution.

Figures 10 and 11 show the time history of the counter-rotating vortex pair for the baseline, and fine lattices, respectively. They clearly show the behaviour of the vortex pair between two and four domain revolutions. The solution is similar on both lattice sizes and so, the baseline lattice is fine enough for a lattice-converged result. The figures show that the vortex pair continues to diffuse over time due to the lattice viscosity used in the calculation. This diffusion also causes the vortex pair to slow down over time.

4.5. Real time flow around a cylinder

For the real-time flow around the cylinder, the lattice size was set to 600×100 with a lattice spacing of 0.01, a lattice velocity of 0.1 and a Reynolds number of 1000. This means that 1000 time steps are required to simulate a second of real time. With the given lattice size, at least 60 MLUPS are required which is possible within a single ARCHER node. It should be noted that this ignores data I/O which is around 25% of the total run time due to 24 flow fields outputted per second for flow visualisation. The full cycle of the shedding is shown in Figure 12. The inlet was free steam, and the boundary layer starts to develop on the upper and lower walls. Due to the vortex shedding and the close proximity of the walls, the cylinder wake interacts with the wall boundary layer. The lattice spacing is enough to resolve the flow features while the lattice speed is high. For this case the equilibrium function was truncated to second order and so terms of the size lattice speed cubed have been dropped.

4.6. Three dimensional flow around the simple frigate shape 2

For this three dimensional case, the top and bottom wall were change from no-slip to slip to remove the boundary layer formation since at the current Reynolds number of 650 most of the vessel would have been contained within the boundary layer. The inflow and outflow boundary conditions where also replaced, with periodic conditions, since the restriction they placed in the Reynolds number caused the flow to become steady. This also allows for a smaller τ to be used, and hence a higher Reynolds number. The third dimension also had periodic boundaries. The last change was to reduce the lattice velocity to 0.07 since the maximum speed in the two dimensional simulation

was around 0.15 which is probably too high for the approximation of the equilibrium equation. This does mean that 30% more time steps are needed for each flow second, and consequently 30% is added to the computational resource required for real-time computations.

The lattice size was $900 \times 100 \times 240$, so the require performance needed to obtain a real time computation is of the order of 30800 MLUPS. With the current performance of just over 130 MLUPS for an ARCHER node we need 240 nodes or some 5760 cores for real-time. While these numbers are feasible they are not within the reach for facilities currently linked to flight simulators. For KNL processors this would be brought down to 130 nodes with the current implementation with the chance of better performance if the Virtual Processing Units (VPU) are utilized.

Results are presented for three cut planes shown in figure 13. The first plane is parallel to the boundary half way up the ship hull. The second cut plane is 66% along the landing deck at the rear of the vessel while the last plane is a cut through the centerline. The results at 15 degrees can be seen in figures 14 for behind the vessel, figure 15 for the cut through deck, and figure 16 for a cut through the centerline of the vessel. The results show many more vortical structures on the starboard side due to the wakes generated by both the bow and superstructure of the vessel. Theses vortical structures are then advected aft. Due to the low Reynolds number and coarseness of the lattice, the flow field above the deck is almost steady but does show a pocket above the deck with very low vorticity levels.

5. Conclusions

This paper presents the details of implementing the LBM method efficiently on several parallel platforms. The main algorithm was re-written to allow for maximum LB updates per second. Additional modifications were put in place to allow for the exploitation of modern CPUs like KNL systems. The proposed coding is both efficient and easy to understand, and stems naturally out of a straightforward LBM implementation. Real-time execution is possible on large computer clusters and the use of KNL opens the gate for linking high performance clusters with real-time wakes in flight simulators.

Future work is currently directed towards furthering the real-time flow capabilities using VPUs and improving the the algorithm. An investigation in using a Hybrid MPI/OpenMP code with dynamic thread scheduling to allow for a more balanced work-load, is also planned.

5.1. Acknowledgments

This work is funded under the Engineering and Physical Sciences Research Council Embedded CSE (EPSRC/eCSE) support grant eCSE05-04 which provides funding to develop software to run on ARCHER and carried out in collaboration with Dr. Gavin Pringle of the EPCC. The use of the UK National Supercomputing Service ARCHER, and the West of Scotland Computing service ARCHIE-WeSt are all gratefully acknowledged.

Barakos, G. N., Woodgate, M. A., 2017. KNL performance comparison: HLBM computer code of Glasgow University. `http://www.archer.ac.uk/community/benchmarks/archer-knl`, accessed: 01-01-2018.

Chapman, S., Cowling, T. G., 1991. The mathematical theory of non-uniform gases: An account of the kinetic theory of viscosity, thermal conduction, and diffusion in gases. Cambridge University Press.

Chen, S., Doolen, G. D., 1998. Lattice Boltzmann method for fluid flows. Annual review of fluid mechanics 30 (1), 329–364.

Chikatamarla, S., Ansumali, S., Karlin, I., 2006. Entropic lattice Boltzmann models for hydrodynamics in three dimensions. Physical review letters 97 (1), 010201.

Crozon, C., Steijl, R., Barakos, G. N., 2014. Numerical study of helicopter rotors in a ship airwake. Journal of Aircraft 51 (6), 1813–1832.

Frisch, U., Hasslacher, B., Pomeau, Y., Apr 1986. Lattice-gas automata for the Navier-Stokes equation. Phys. Rev. Lett. 56, 1505–1508.

Horn, J. F., Bridges, D. O., Wachspress, D. A., Rani, S. L., 2006. Implementation of a free-vortex wake model in real-time simulation of rotorcraft. Journal of Aerospace Computing, Information, and Communication 3 (3), 93–107.

Karlin, I., Ferrante, A., Öttinger, H., 1999. Perfect entropy functions of the lattice Boltzmann method. EPL (Europhysics Letters) 47 (2), 182.

Khan, M. A. I., Delbosc, N., Noakes, C. J., Summers, J., 2015. Real-time flow simulation of indoor environments using lattice Boltzmann method. Building Simulation 8 (4), 405–414.

Latt, J., 2007. How to implement your DdQq dynamics with only q variables per node (instead of 2q), Technical Report. Tufts University, Medford, USA.

Malaspinas, O., Sagaut, P., 2012. Consistent subgrid scale modelling for lattice Boltzmann methods. Journal of Fluid Mechanics 700, 514–542.

Mattila, K., Hyvluoma, J., Timonen, J., Rossi, T., 2008. Comparison of implementations of the lattice-Boltzmann method. Computers & Mathematics with Applications 55 (7), 1514 – 1524.

Openlb, 2016. Openlb open source lattice Boltzmann code. `http://www.optilb.org/openlb`, accessed: 01-07-2016.

Palabos, 2016. Palabos an open-source CFD solver based on the lattice Boltzmann method. `http://www.palabos.org/`, accessed: 01-07-2016.

Spence, G. T., Moigne, A. L., Allerton, D. J., Qin, N., 2007. Wake vortex model for real-time flight simulation based on large eddy simulation. Journal of aircraft 44 (2), 467–475.

Succi, S., Karlin, I. V., Chen, H., 2002. Colloquium: Role of the H theorem in lattice Boltzmann hydrodynamic simulations. Reviews of Modern Physics 74 (4), 1203–1220.

Wellein, G., Zeiser, T., Hager, G., Donath, S., 2006. On the single processor performance of simple lattice Boltzmann kernels. Computers & Fluids 35 (89), 910 – 919.

Zou, Q., He, X., 1997. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. Physics of Fluids 9 (6), 1591–1598.

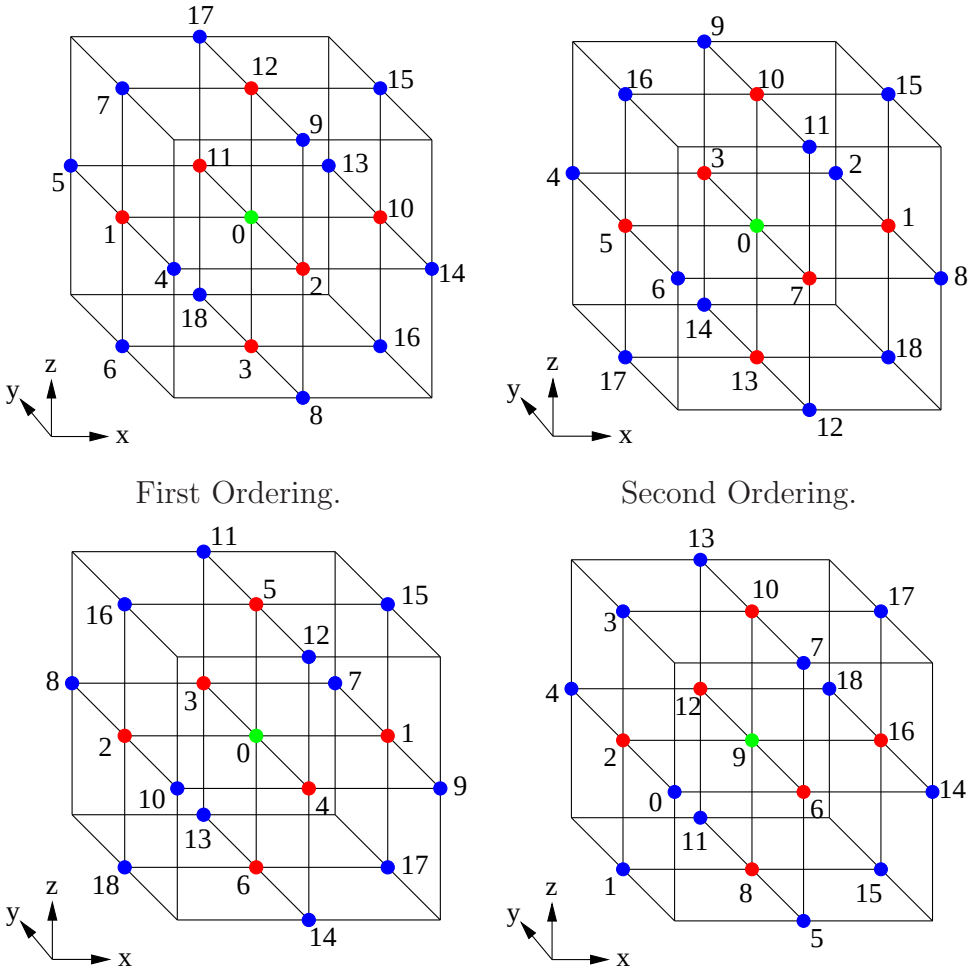
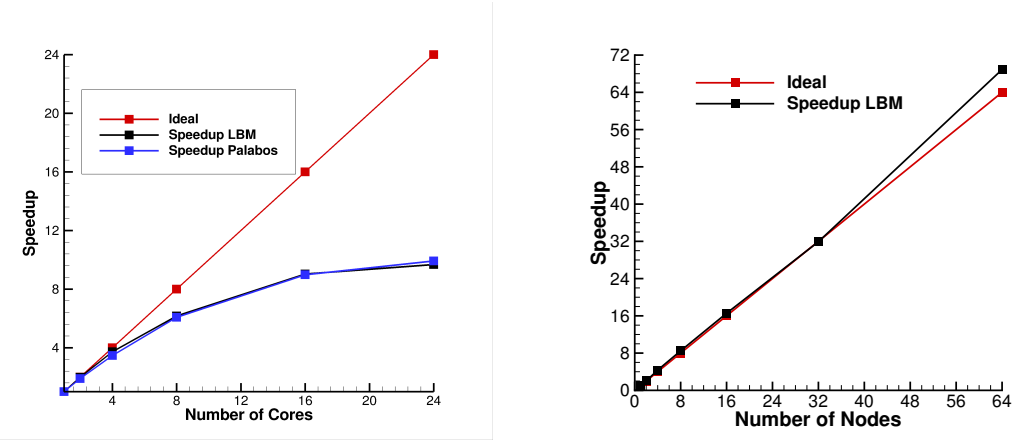


Fig. 4 – Four possible orderings of the D3Q19 lattice nodes.



(a) Performance within a compute node (b) Performance across compute nodes

Fig. 5 – The speedup curve for running LBM within and across ARCHER compute nodes - (Two 2.7GHz 12-core E5-2697 v2 Processors)

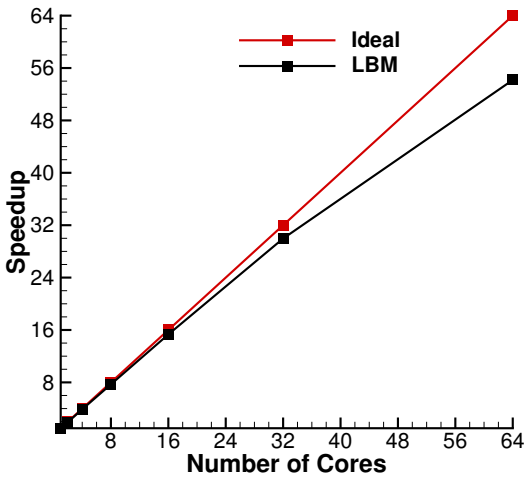


Fig. 6 – The speedup curve for running LBM within a 64-core KNL processors (model 7210) running at 1.3Ghz.

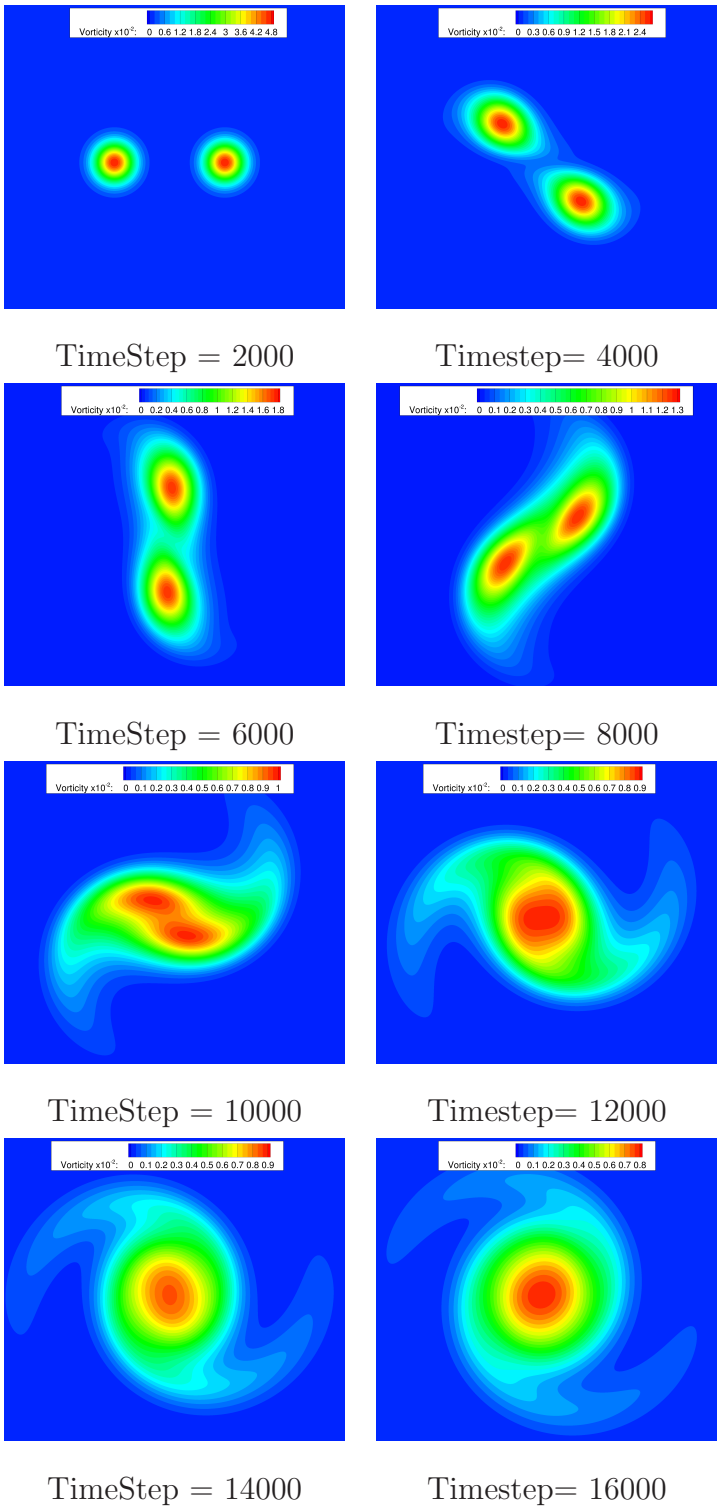
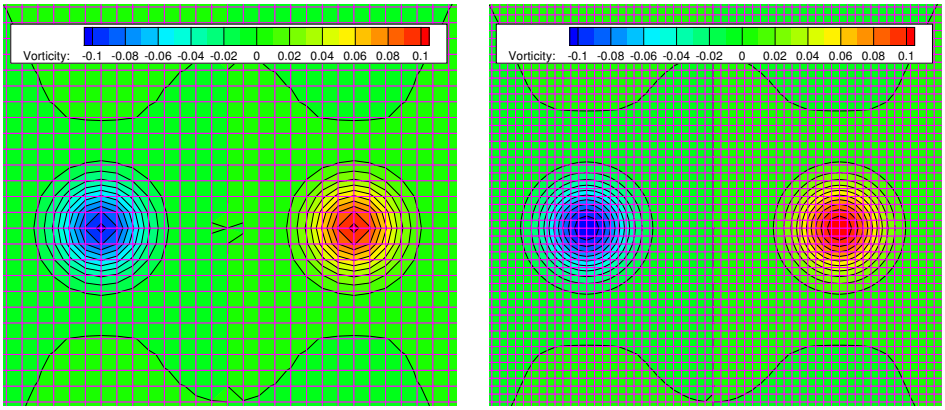
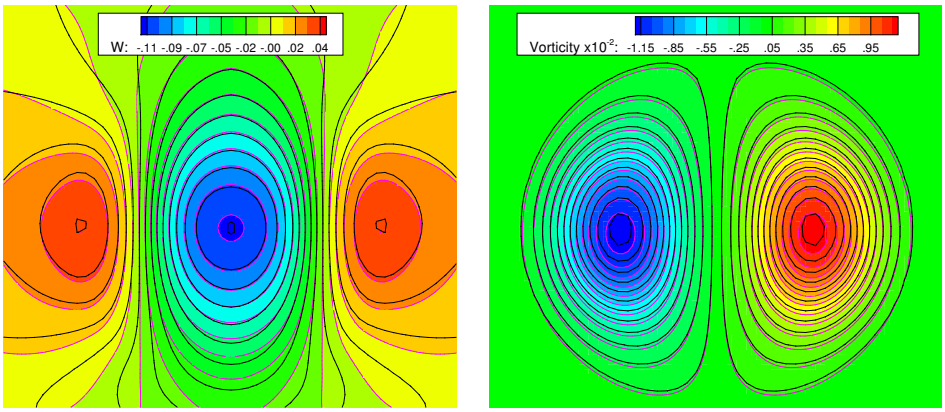


Fig. 7 – The merging of a Co-rotating Vortex Pair on a 801×801 with lattice viscosity of 0.01



Baseline Lattice 101×201 Fine Lattice 201×401

Fig. 8 – The initial vorticity on the baseline and fine meshes for the Counter-rotating Vortex Pair



Downward W Velocity Vorticity

Fig. 9 – A comparison between velocity and vorticity of the baseline, black contours, and fine lattice, purple contours, for the Counter-rotating Vortex Pair after four loops

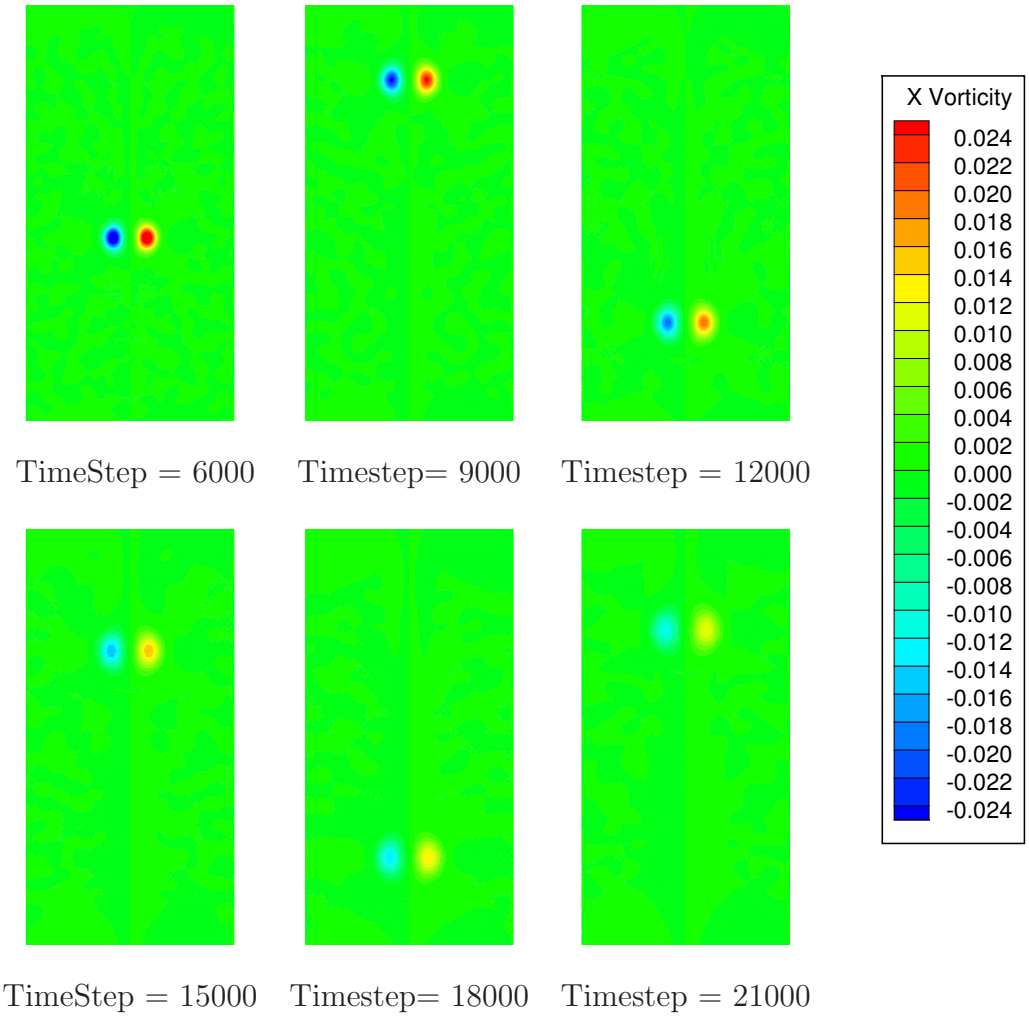


Fig. 10 – The vorticity of the baseline 101×201 lattice for the Counter-rotating Vortex Pair

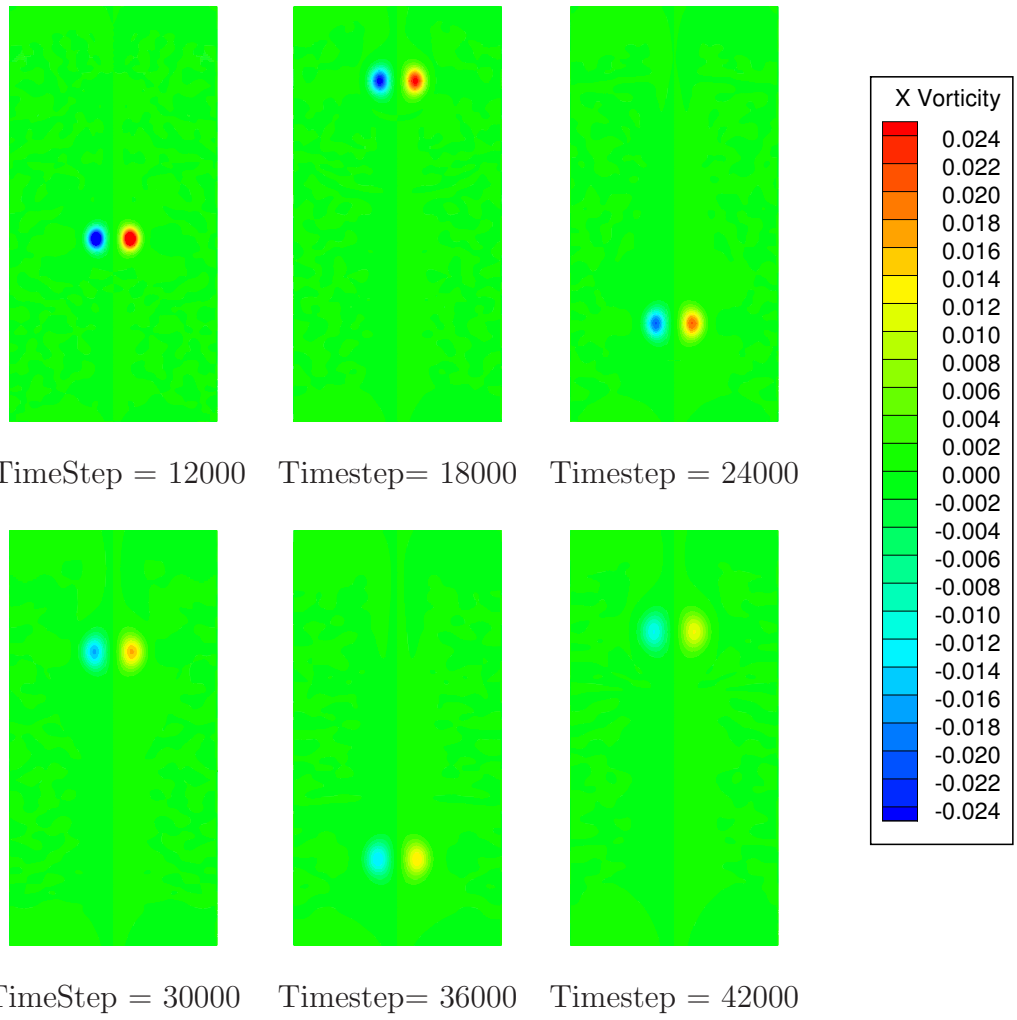
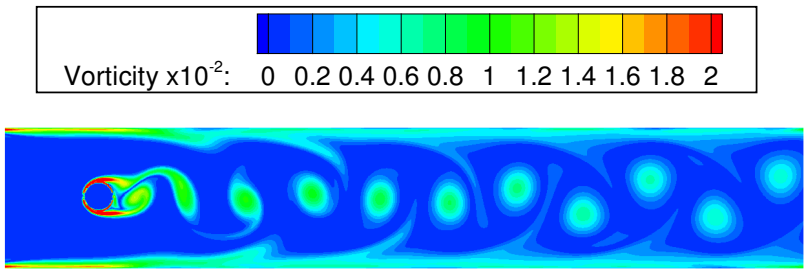


Fig. 11 – The vorticity of the fine 201×401 lattice for the Counter-rotating Vortex Pair



(a) Timestep = 65000

(b) Timestep = 65500

(c) Timestep = 66000

(d) Timestep = 66500

Fig. 12 – The vorticity magnitude for flow around a cylinder within a channel

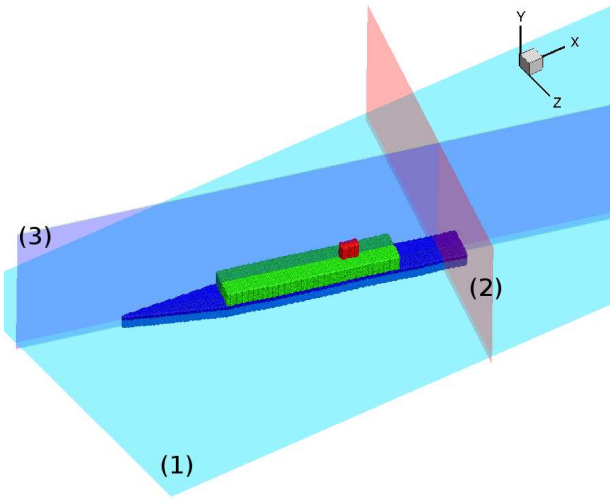


Fig. 13 – Position of the three cut planes with respect to the SFS2 geometry.

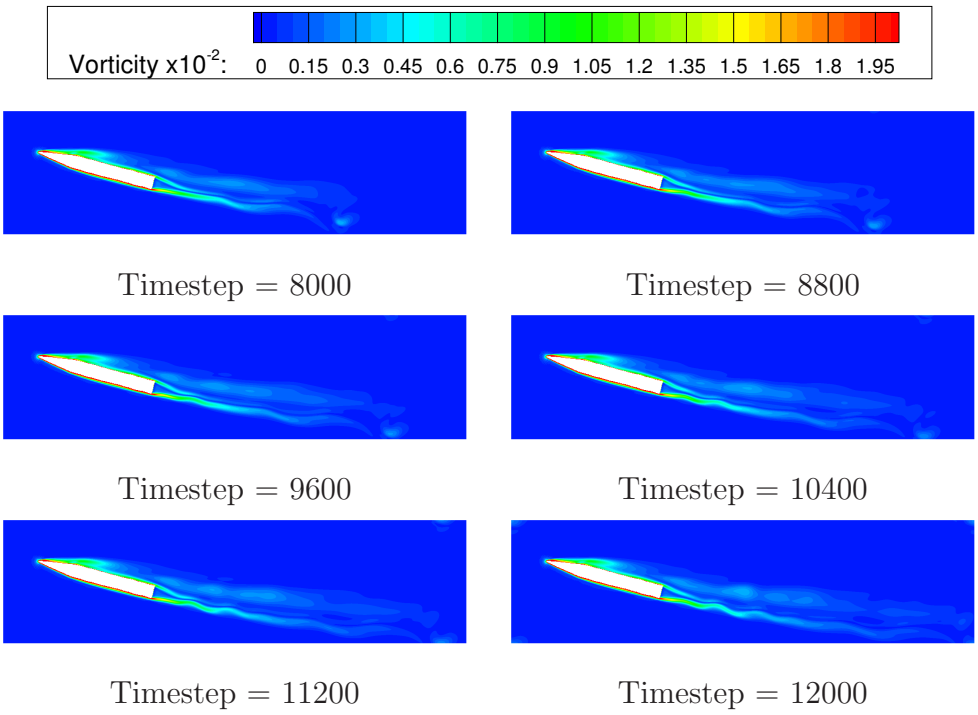


Fig. 14 – The vorticity magnitude for flow around the SFS2 at 15 degrees on the first cut plane.

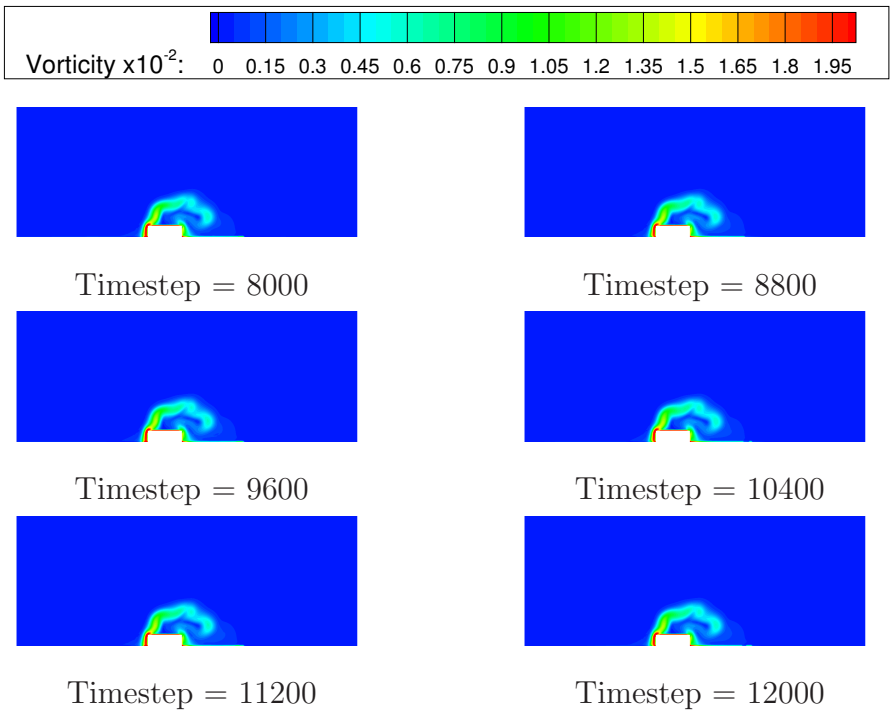


Fig. 15 – The vorticity magnitude for flow around the SFS2 at 15 degrees on the second cut plane.

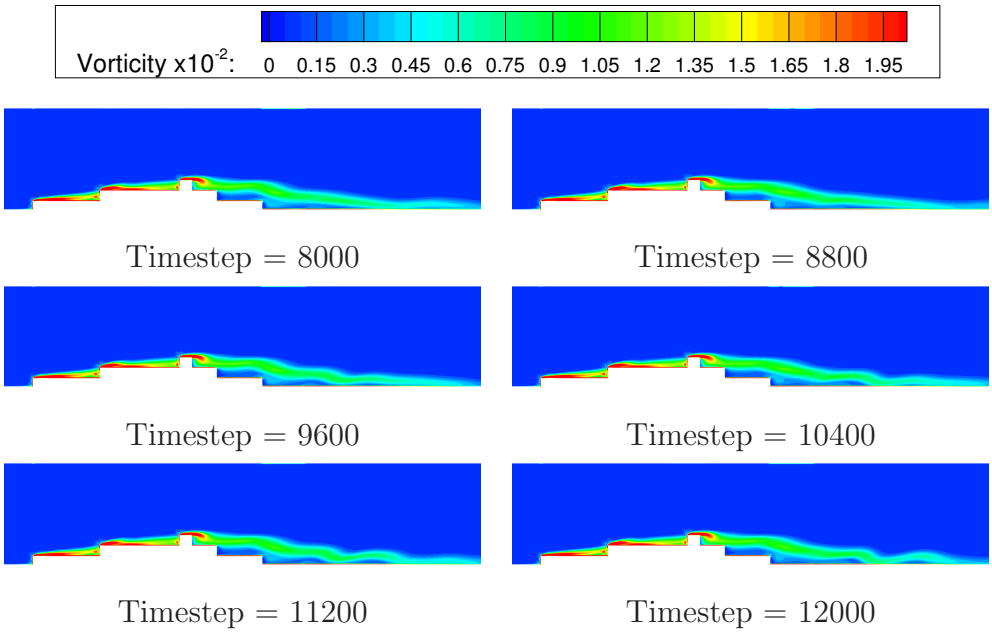


Fig. 16 – The vorticity magnitude for flow around the SFS2 at 15 degrees on the third cut plane.